CHAPTER 1

INTRODUCTION

1.1 Introduction

The performance of modern microprocessors is increasingly dependent on their ability to execute multiple instructions per cycle. Such rapid, dramatic increases in hardware parallelism have placed tremendous pressure on compiler technology. For years, a steadily growing clock speed has been relied upon to consistently deliver increased performance for a wide range of applications. Recently, however, this trend has changed, as the microprocessor industry can no longer increase clock speed because of difficulties related to power consumption, heat dissipation, and other factors. Meanwhile, the exponential growth in transistor count remains strong, causing major microprocessor companies to add value by producing chips that incorporate multiple processors.[4] To achieve very high-performance of processors the computer architects must concern on cost, time, speed variables to follow the micro-processors trends. Thus to achieve low time computation with high speed performance and with low cost processor, the computer architects need to deal with cache memory hierarchies and exploit instruction level parallelism.[5]

The continuing trend of microprocessors, the increasing gap between memory speed and the processor speed necessitates new techniques for memory latency tolerance. To develop these techniques, a high-level understanding of the memory characteristics of programs is required. This is to understand how programmer intended

to use the memory, not just how the individual load/store operations in the program behave. [3] Current microprocessors spend a large percentage of execution time on memory access stalls, even with large on-chip caches. Since processor speeds are growing at a greater rate than memory speeds, the expectation of memory access costs to become even more important in the future. Figure 1.1 shows the graph performance vs. time of Processor and DRAM by Moore's Law and the gap between processor-memory speeds grows 50% per year from year 1980 until 2000.

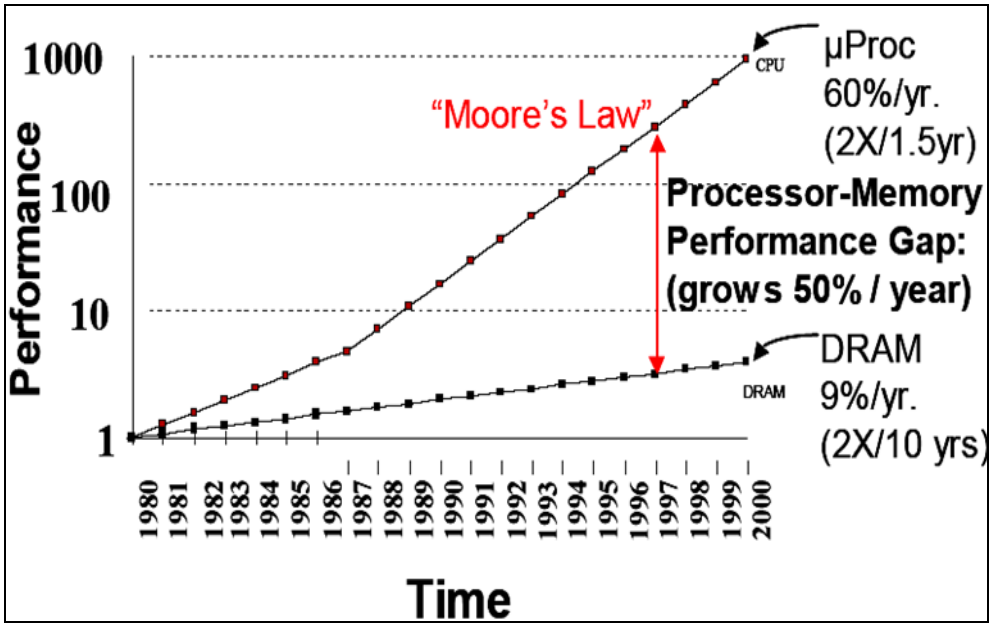

Figure 1.1: Processor-DRAM Memory Gap (latency) by Moore's Law

Refer to the graph, the microprocessor performance increase 60% per year while memory performance increase only 9% per year. Computer architects have been battling this memory latency problem by designing ever larger and more sophisticated caches. Although caches are extremely effective, they are not the complete solution. Other techniques are required to fully address the memory latency problem. [2] Memory latency problem is a problem due to the gap between CPU speed and memory speed, where CPU speed continue to growth while memory doesn't. This problem happens when CPU access to the main memory, where CPU speed is high contrast with memory speed is slow. Then CPU need to deals with cache memory, but it is still have a problem that will cause cache-miss problem. This is why cache memory is not a complete

solution for tolerating memory latency problems. As the performance difference between the CPU and the main memory increases, reduction of the cache misses and penalties become more severe.

One of the techniques to reduce cache misses is to prefetch data or instruction. Prefetch by definition is to fetch data or instruction before they are requested by the processor. This prefetch can be done by prefetching techniques. Prefetching techniques can be performed either by hardware and/or by software. Hardware can be designed to prefetch instructions and data, either directly into the cache or into an external buffer that can be more quickly accessed than main memory. On the other hand, software prefetching is implemented by including fetch instructions in processor instruction set. Fetch instructions can be coded explicitly by the programmer or added by the compiler during the optimization. [6] In both software and hardware prefetching, the mechanisms is based on overlapping execution by the prefetching of instructions or data.

Instruction prefetching speculatively brings the instructions needed in the future close to the microprocessor and, hence, reduces the transfer delay due to the relatively slow memory system. If instruction prefetching can predict future instructions accurately and bring them in advance, most of the delay due to the memory system can be eliminated. [1] Data prefetching is a technique for hiding the access latency of data referencing patterns that defeat caching strategies. Rather than waiting for a cache miss to initiate a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. To be effective, prefetching must be implemented in such a way that prefetches are timely, useful, and introduce little overhead. Secondary effects such as cache pollution and increased memory bandwidth requirements must also be taken into consideration. Despite these obstacles prefetching has the potential to significantly improve overall program execution time by overlapping computation with memory accesses. [17]

1.2 Problem Background

Memory latency becoming an increasing important performance bottleneck as the gap between processor and memory speeds continues to grow. While cache hierarchies are an important step toward addressing the latency problem but they are not a complete solution. To further reduce or tolerate memory latency problem, several techniques have been proposed and evaluated which is responsible to reducing memory latency by cache-misses. Regarding to this problem, it has become necessary for us to have a better compiler optimization techniques. One of the techniques has recently used was Software Prefetching. Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run time, the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. Software prefetching has been shown to be effective in reducing memory stalls in array-based applications for both sequential and parallel applications, particularly for scientific programs making regular memory accesses. This make prefetching has enjoyed considerable success for array-based applications but its potential in pointer-based applications has remained largely unexplored [7]. Most of the commercial applications, such as database engines, of-ten use hash tables and to trees represents and store data. These structures used pointer-based such as linked-list data structure that are often traversed in loops or by recursion. This linked-list data structure also known as Recursive Data Structures.

Recursive Data Structures (RDSs) include familiar objects such as linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure. Recursive data structures can be broadly interpreted to include most pointer-linked data structures (e.g., mutually-recursive data structures, or even a graph of heterogeneous objects). Recursive data structures are one of the most common and convenient methods of building large data structures (e.g, B-trees in database applications, octrees in graphics applications, etc.). From a memory performance perspective, these pointer-based data structures are

expected to be suffer a large memory penalty due to data replacement misses, temporal locality may be poor when traversal a large RDS and little inherent spatial locality between consecutively-accessed nodes in an RDS. Therefore, techniques for coping with the latency of accessing these pointer-based data structures are essential. [3,7,9,10]

Prefetching for pointer-based data structures is challenging due to the memory serialization effects associated with traversing pointer structures. The memory operations performed for array traversal can issue in parallel because individual array elements can be referenced independently. At worst, pairs of array references are serialized in the case of indexed array traversal. But even in that case, separate indexed array references can perform in parallel. In contrast, the memory operations performed for pointer traversal must dereference a series of pointers sequentially. [2,7] The memory serialization in pointer chasing prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain, thus limiting their effectiveness. This property forces associated memory references to be sequentialized, and is known as the pointer-chasing problem.

Pointer-chasing applications usually exist in programs solving complex problems where the amount and organization of data is unknown at compile time, requiring the use of pointers to manage both dynamic storage and linkage. They may also arise from high-level programming language constructs such as object-oriented programming. Because memory is allocated and accessed dynamically, the access pattern tends to be very irregular and lack locality, resulting in poor cache performance. [2]

1.3 Problem Statement

Previous techniques of software prefetching for pointer based codes influence the processor performance and accuracy of prediction prefetch instructions. This project examines the question

How to reduce or tolerate memory latency by using L1-cache-miss in pointer-based codes?

Today's microprocessor performance deals with ILP and cache memory to achieve highest performance. However, most of today's applications are very complex and the processor performance becomes slow. This may due to the pointer-based data structure used in the applications. This project also explores the sub questions that are:

i. *How to exploit parallelism in processor? Is that by fetch all the prefetch instructions?*
ii. *What are the latency variables that make prefetching algorithms critical?*
iii. *What types of pointer-based codes that makes memory performance becomes very slow?*

1.4 Project Objectives

The objectives of the project are:

i. To investigate, experiment, compare and choose the best critical latency variables from the existing software prefetching techniques of pointer based codes.
ii. To design and develop the propose pointer prefetching algorithm using the chosen critical latency variables.
iii. To test and implement the propose pointer prefetching algorithm applying to the compiler for program containing RDS.

1.5 Project Scopes

The scopes of the project are:

i. The comparative study of previous Prefetching techniques only for Greedy Prefetching technique, Jump History Pointer technique and Prefetch Array technique.
ii. Focus only the cache misses in Level-1 cache.
iii. Develop the prefetching algorithm using C programming.
iv. Using data library from benchmark suite those containing pointer-based data structures also known as Recursive Data Structures.
v. Using three Olden benchmark programs that are mst, health and perimeter that classified as tree and list traversal to evaluate the compiler performance where it contents different types of Recursive Data Structure.
vi. Simulation of these compiler techniques will be simulate on dynamically-scheduled, superscalar processor similar to SPARC using Simics version 3.0.

1.6 Project Contributions

This project will give better insights and idea or solution to expand the compiler's scope to include another important class of applications: those containing pointer-based data structures also known as Recursive Data Structures. Proposing a better algorithm for pointer-based codes will give another opportunity for compiler technology to develop an effective and optimize for today compiler. The comparative study on previous techniques will help the understanding on the compiler improvements and problems.

1.7 Conclusion

Nowadays, the applications becomes larger compared than recent years where only consists of small programs and execute sequentially is necessary. Compared than larger applications such as B-trees in database applications, oc-trees in graphic applications where it suffered for large memory penalty due to data replacement misses and consecutive elements is not at contiguous address. One of the most common and convenient methods of building large data structures is Recursive Data Structures. Due to these large applications, the execution speed is low because of pointer-chased problem and the disparity gap between the CPU speed and memory speed. To overcome the read latency, this project will propose new algorithm for compiler-based Prefetching technique and compare with previous technique to give the best result for improvement execution speed in superscalar microprocessor.